**BLG 431E SYSTEM PROGRAMMING**

**PROJECT 1**

**Group 57**

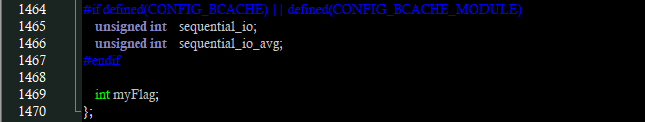
**Ahmet Göktuğ SEVİNÇ – 150140120**

**Abdullah Melih Canal – 150130056**

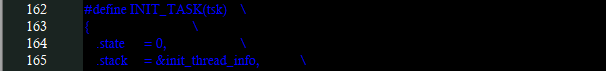
In the project we were asked to implement a new system call (*set\_myFlag*) to set the value of a flag (*myFlag)* which is added into task descriptor of a process. To be able to add a new flag into the processes we had to change the task descriptor of the processes. In Linux, the kernel stores the list of processes in a circular doubly link list called the *task list*. Each element in the task list is a *process descriptor*  which contains all of the information about a process and these process descriptors are the type of *struct task\_struct*. Therefore, firstly, we added a new field into that *task\_struct* structure which is defined in *<linux/sched.h>*  and that header file was located in */include/linux/shed.h* path :



**...**



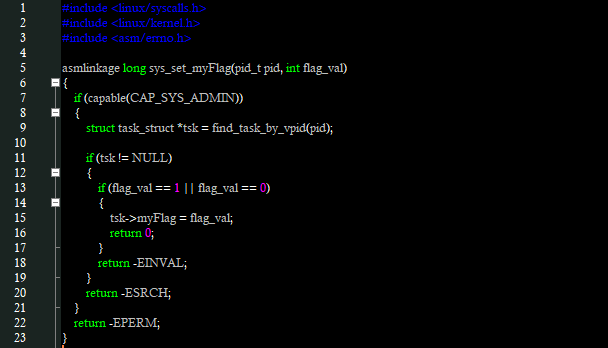
After adding our flag to the task descriptor, to initialize our new field during system initialization (creation of process 0) we needed to update INIT\_TASK macro located in */include/linux/init\_task.h* :



**...**



After those initial operations, we wrote our system call function and added it to the kernel. For this purpose, we created a new folder (*set\_myFlag*) under root of the source and in that folder we created our new system call file (*set\_myFlag.c*) :



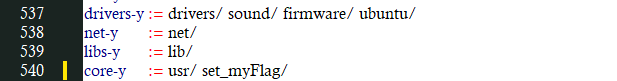
First of this system call can only be used by processes with root privilages so first *if* statement ensures that restriction. If the calling process does not have the root privilages we returned an error (*-EPERM*) to indicate that it is not permitted to do that operation. Then to be able to obtain a pointer to the task descriptor of a process with its pid we examined *sched.h* file and found the appropriate function *find\_task\_by\_vpid()* which takes a pid as a parameter. Then we simply checked whether we obtained the pointer or not. If not, we returned an error (*-ESRCH*) to indicate a process with that pid could not be found. And finally, we checked the given flag value to the function. If it is not zero (0) or one (1), we returned another error (*-EINVAL*) to indicate value is invalid. If all constraints are satisfied, we assigned flag value to the current process.

After writing our system call, we also created a *Makefile* under that folder and added it to the Makefile of the source to make this call available at the compilation.

/set\_myFlag/Makefile:

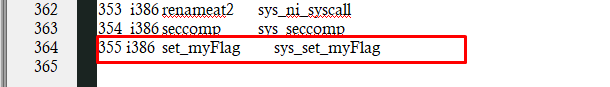


/Makefile:

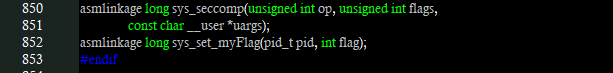


Then we modified system call table under */arch/x86/syscalls/syscall\_32.tbl* and system call header file under */include/linux/syscall.h*.

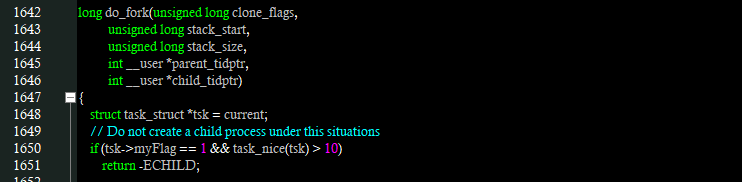
*/arch/x86/syscalls/syscall\_32.tbl:*



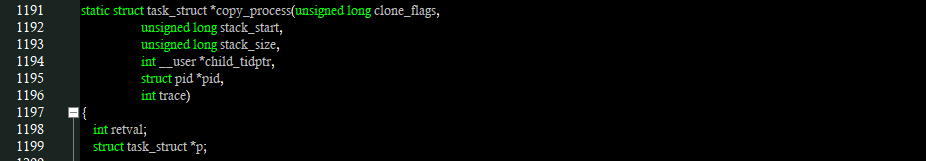
*/include/linux/syscall.h:*



Now, since all of the operations regarding system call was finished, we could start to second requirement of the project: modifying fork and exit system calls. In Linux, new processes are created by *fork* system call and that call uses *do\_fork* function located in */kernel/fork.c*. So, we were asked to change that function so that, if the value of our new flag (myFlag) is zero (0), default actions will be performed but it is one (1) and its nice value is greater than 10 no processes will be created.



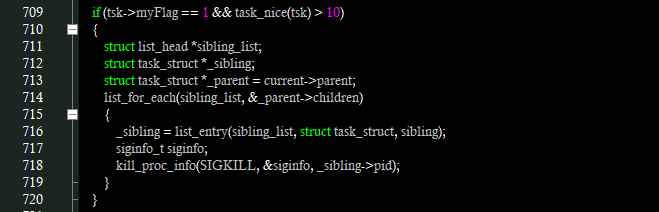
Here, inside do\_fork function, first of all we read the current process and checked its flag value and nice value. If its flag is 1 and nice value is greater then 10, we returned from the function with an error (*-ECHILD*) to indicate no child process will be created. If these conditions are satisfied, do\_fork function calls another function *copy\_process* to create new child process. Therefore, we simply added a line to initialize the flag of the child processes.



**...**



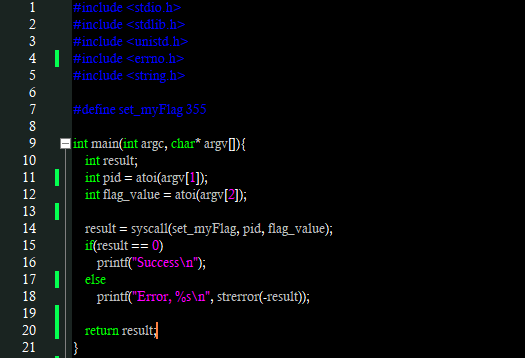
Finally, we were also asked to modify exit system call so that, if the myFlag value of process is zero (0), default actions will be performed, but if its value is one (1) and its nice value is greater than 10, not only that specific process, but all of its siblings will also be terminated. In linux processes are terminated via exit system call and for this purpose *do\_exit* function is used located in */kernel/exit.c* file. So, we made some changes in that function.



Here, first of we checked whether the flag value of the current process is 1 and its nice value is greater than 10, or not. If these conditions were satisfied, we needed to kill all of the siblings of that process along with itself. To be able to do that, we used *list\_for\_each* and *list\_entry* macros. Firstly we created a linked list called *sibling\_list,* then we created two task\_struct pointers to point each of siblings (*\_sibling*) and the parent of the current process (*\_parent*). After that we used *list\_for\_each* macro to iterate over all of the childrens of that parent and by using *list\_entry* macro we obtained those child processes and terminated them by sending *SIGKILL* signal.

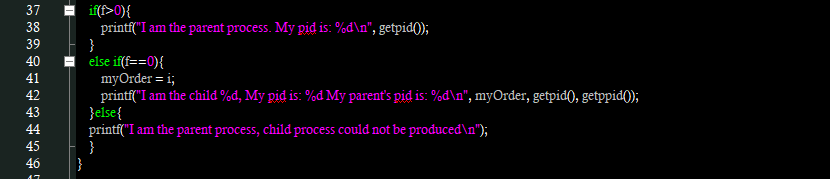
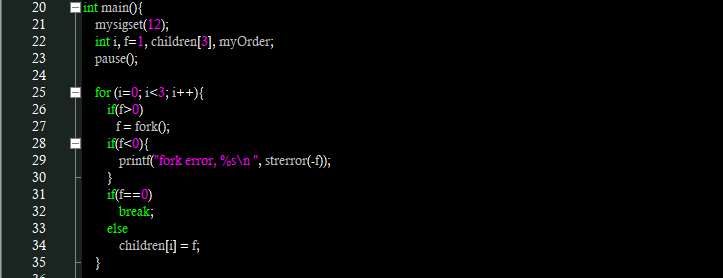
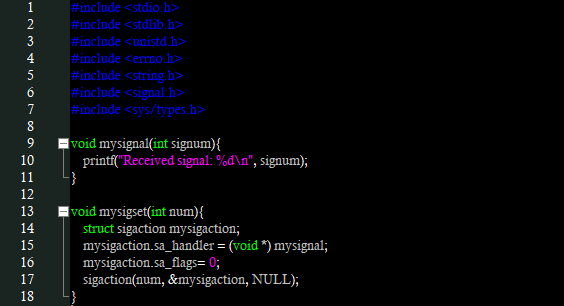
**TESTING**

For test operations we wrote 3 fonctions. Firstly, we wrote a function to set the flag (my\_Flag) value of a process from the command line. *set\_Flag.c* :

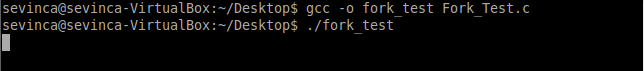


This function takes process id, and the flag value from the command line. Then sets the flag of the process by calling our system call and prints the result.

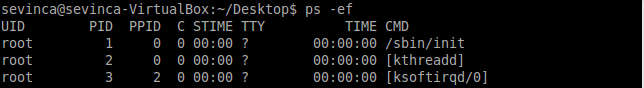
To test *fork()* operations we wrote another function. *Fork\_Test()*:



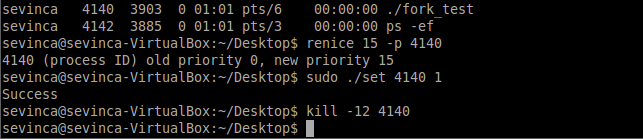
At the beginning of that function we call our sigset functun with the signal number of 12 and then we immidiately pause the process. If we send kill signal from the command line, everthing will work properly and there will be 1 parent and 3 child processes and they will print their information as output. But, if we set flag of our process to 1 and set the nice value of it to above 10, no child will be produced.



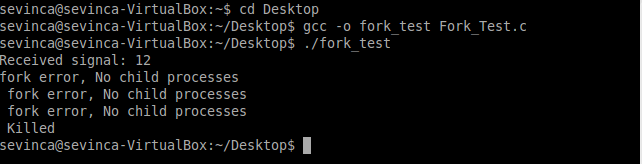
Here we started our function and it is paused. Waiting for a signal. Now we can check process list and find the process number of that process. To do this we used *ps –ef* command that lists current processes.



...

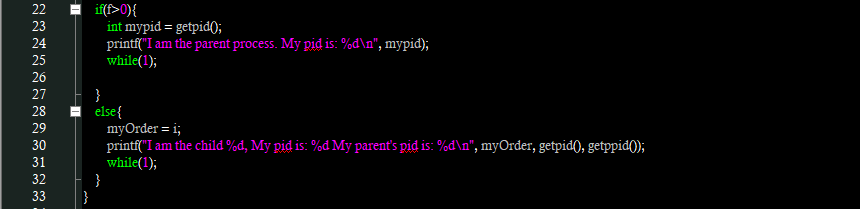
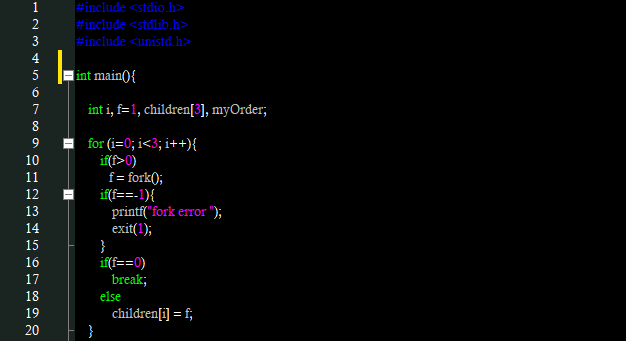


After learning our process number, we set our flag value to one and nice value of our process to 15, by using *renice* program. Then, we sent kill signal(12) to the process.

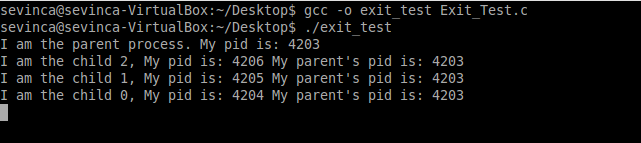


As expected, no child process is produced.

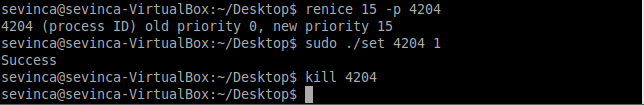
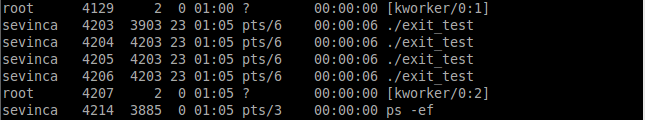
To test our exit system call we wrote another function. *Exit\_Test.c* :



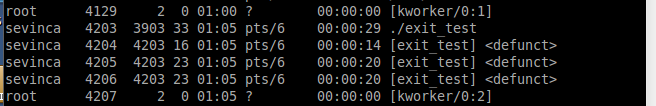
Here, we simply create 1 parent and 3 child processes and put all of them into infinite while loop.



As it is seen, 1 parent and 3 child processes are created. Now since we know process numbers of our child processes, we can set one of their flag to 1 and nice value to greater than 10. But, before doing that we checked the current process list, by using *ps –ef* command again.



Here, we set the nice value and flag of a child process with the process number of 4204. Then we killed that child process by sending *–kill* signal to it. Then, we checked process list again.



As expected, not only the process with the pid of 4204, but also its siblings (4205, 4206) are also killed. As it can be seen in above images, before using out *set\_Flag* function we used *–sudo* command to get root priveleges otherwise we got an error saying we don’t have permission for that operation.